Cruzeiro do Sul
Educacional

JavaScript

Conteudista
Prof. Me. Marco Antonio Sanches Anastacio

Revisão Textual
Prof.ª Dra. Selma Aparecida Cesarin

OBJETIVOS DA UNIDADE

- Entender as regras, a função e a importância da Linguagem de Programação *Javascript* na construção de páginas *web*;
- Compreender os conceitos de eventos, comandos e funções, aplicando os conhecimentos adquiridos na construção de páginas interativas.

Atenção, estudante! Aqui, reforçamos o acesso ao conteúdo *on-line* para que você assista à videoaula. Será muito importante para o entendimento do conteúdo.

Este arquivo PDF contém o mesmo conteúdo visto *on-line*. Sua disponibilização é para consulta *off-line* e possibilidade de impressão. No entanto, recomendamos que acesse o conteúdo *on-line* para melhor aproveitamento.

Introdução ao *Javascript*

Em maio de 2022, uma pesquisa realizada pelo *StackOverFlow* com mais de 70.000 desenvolvedores revelou como aprendem e sobem de nível, quais ferramentas estão usando e o que desejam. No topo das Linguagens de Programação mais utilizadas, encontramos o *JavaScript*, com mais de 65% de preferência, o que mostra o porquê de ser considerada a **linguagem** mais popular entre os desenvolvedores.

Site

Visite a página do *StackOverFlow* e conheça mais detalhes dessa pesquisa.

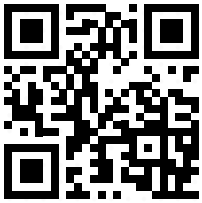
Mas, afinal, o que é e para que serve o *JavaScript*?

O *JavaScript* (ou simplesmente *JS*) é uma Linguagem interpretada, de alto nível, para programação de uso geral, muito aplicada no desenvolvimento *web*. Conhecida como a Linguagem da *Internet*, já que é nativa dos navegadores, o JS permite realizar validações do lado cliente e interações entre páginas, além de alterar dinamicamente estilos e elementos da página HTML.

Muito conhecido por sua capacidade de lidar com o processamento do lado do cliente (*front-end*), o JS evoluiu para o lado do servidor (*NodeJS*) e, hoje, pode ser utilizado, inclusive, no desenvolvimento híbrido para dispositivos móveis, por meio do *framework Apache Cordova*.

Site

Visite a página do *Apache Cordova* para saber mais sobre desenvolvimento *mobile* multiplataforma com o HTML, CSS e JS.



O uso do *JavaScript* permite a criação de efeitos nas páginas *Web*, validação de formulários, identificação do contexto e definição de comportamentos em função desse contexto, entre outros. No lado do usuário, o navegador é o responsável pela interpretação das instruções dos *scripts*, executando-as de modo a permitir essa interatividade.

Todo o Código *JavaScript* é executado somente quando o evento ao qual está associado é acionado e, por esse motivo, o JS usa o chamado "modelo de execução controlado por eventos". Porém, em alguns casos, podemos inserir o Código sem a associação a eventos e, dessa forma, a instrução ou o comando será executado conforme o navegador interpreta a página *web*.

As principais vantagens do *JavaScript* são:

- **Linguagem leve:** o interpretador é nativo dos navegadores, o que permite que o código seja executado na sua forma natural e rápida, possibilitando que alterações e erros sejam tratados em tempo de execução;

- **Flexibilidade e versatilidade:** por ser uma Linguagem nativa dos navegadores, o processo de desenvolvimento *web* é simplificado. Mas não é somente disso que estamos falando! Hoje, já é possível executar o *JavaScript* no lado do servidor, além de sua aplicação para criar jogos, aplicativos *desktop* e até *mobile*;

- **Comunidade consolidada:** por ser umas das Linguagens mais conhecidas do mundo, sua comunidade é também outro ponto forte, o que significa ajuda disponível em diversos canais de comunicação, farta disponibilidade de Bibliotecas e ou *frameworks*, que contribuem muito para o crescimento da Linguagem.

Iniciando com *JavaScript*

Podemos inserir um *script* JS no documento HTML das seguintes formas:

Dentro de uma *tag* HTML;

- ***JavaScript in-line*:** o *script* é inserido no corpo da página **<body></body>**;

- ***JavaScript* interno:** fazemos a declaração na seção **<head></head>**;

- ***JavaScript* Externo:** por meio de um arquivo que tem a extensão **.js**.

Veremos, a seguir, como utilizar cada um desses casos e alguns exemplos de aplicação.

JavaScript em uma *Tag*

Quando inserimos um código *JavaScript* em uma *tag*, ele é sempre associado a

um evento. Os eventos serão vistos mais adiante. Por enquanto, faremos um exemplo simples com o evento ***onclick***, que é acionado quando o usuário clica no elemento:

```
<html>
<head>
<title>Exemplo JavaScript</title>
</head>
<body>
<input type="button" value="Clique aqui" onclick="alert('Olá Mundo JS')">
</body>
</html>
```

Importante

Fique atento(a) ao uso de aspas duplas ("...") e simples ('...') e, sempre que possível, evite o uso de eventos direto em *tags* HTML.

JavaScript dentro do *body* e do *head*

Se fizermos uma comparação com CSS, esse modo de inserir *JavaScript* em uma página seria o modo incorporado. Seja dentro do ***body*** ou do ***head***, sempre devemos utilizar a *tag* **<script>...<\script>**, para demarcar o início e o fim do *script* JS. Note que os comandos podem ser executados sem a necessidade de eventos. Isso não é o mais comum, mas é um recurso que usaremos em nossos estudos.

Veja o exemplo a seguir:

```
<html>
<head>
<title>Exemplo JavaScript</title>
<script>
   alert("Eu estou no cabeçalho.");
</script>
</head>
<body>
<script>
   document.write("<b class= 'teste'>Eu estou no corpo do documen-
to.</b>");
</script>
</body>
</html>
```

JavaScript Externo

Assim como em CSS, também podemos criar um arquivo separado do HTML com nossos códigos em *JavaScript*. Esse arquivo deve ser salvo com a extensão **.js** e é chamado no cabeçalho da página com a *tag* **<script>**.

No próximo exemplo, vamos utilizar dois arquivos: **exemplo3.html** e **script3.js**:

```
<!--arquivo exemplo3.html-->
<html>
<head>
<title>Exemplo JavaScript</title>
<script src="script3.js"></script>
</head>
<body>
Conteúdo da página. Também podemos fazer aqui chamadas para
blocos de códigos do arquivo .js.
</body>
</html>
```

```
//arquivo script3.js
alert("Estou em um arquivo .js");
```

Importante

Perceba que no arquivo .js não utilizamos as *tags* **<script>... <\script>** para demarcar o código JS.

O que é um Evento?

Sempre que ocorre uma interação com um documento ou página, um evento é disparado. Um evento pode ser qualquer interatividade do usuário com um elemento HTML e alguns eventos também podem ser disparados pelo navegador.

Alguns exemplos de eventos HTML são:

- Uma página da *web* HTML terminou de carregar;
- Um campo de entrada HTML foi alterado;
- Um botão HTML foi clicado.

As reações aos eventos (programadas em *JavaScript*) são sempre registradas como propriedades dos elementos. O Quadro 1 apresenta uma lista de alguns eventos HTML mais comuns.

Quadro 1 – Alguns eventos HTML

Evento	Descrição
onchange	Um elemento HTML foi alterado.
onclick	Um elemento HTML foi clicado.
onmouseover	O mouse move-se sobre um elemento HTML.
onmouseout	O mouse move-se para longe de um elemento HTML
onkeydown	Uma tecla do teclado é pressionada.
onload	O navegador terminou de carregar a página.

Fonte: Elaborado pelo Conteudista

Saiba Mais

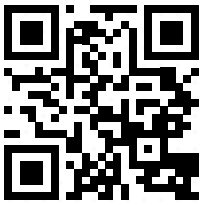
Os manipuladores de evento (***event handler***) são funções a serem executadas quando um evento é acionado ou ocorre uma ação do navegador. E o disparador de evento é o elemento HTML em que o manipulador de evento foi adicionado.

O exemplo a seguir mostra as informações da data a partir do clique no botão:

```
<html>
<body>
<button onclick = "document.querySelector('#data').innerHTML = Date()"> Exibir data </button>
<p id="data"></p>
</body>
</html>
```

Site

O método **querySelector('#data')** retorna o primeiro elemento filho que corresponda ao **id = "data"**.



Leitura

O **innerHTML** é uma propriedade do ***Element*** que define ou retorna o conteúdo HTML de um elemento.

Manipulando Dados com JS

Todos os dados que são manipulados por um Programa são armazenados na memória do computador e podem ser de dois tipos:

- **Constante:** quando o conteúdo não muda durante a execução do programa;
- **Variável:** quando o conteúdo muda durante a execução do Programa.

O *JavaScript* é uma Linguagem de tipagem dinâmica, pois, ao contrário de outras Linguagens que exigem a declaração de uma variável com um tipo de dado definido, em JS, as variáveis não estão associadas diretamente a um tipo específico:

```
let x = 42; // x é do tipo numérico (int)
    x = "bar"; // x agora é um texto (string)
    x = true; // x agora é um valor lógico (booleano)
```

A definição de variáveis exige atenção em algumas regras:

- Toda variável deve começar com uma letra ou um underscore("_");
- Caracteres subsequentes devem ser letras ou números;

- Não deve conter espaço em branco ou caracteres especiais;
- Não deve ser uma palavra reservada.

Além disso, é importante destacar que o *JavaScript* é ***case-sensitive***, e isso significa dizer que todas as variáveis listadas a seguir são diferentes:

- quantidade;
- QUANTIDADE;
- Quantidade;
- QuAnTidAdE.

Existem duas palavras-chave para criarmos variáveis: **var** (antigo) ou **let** (moderno). Até a especificação ECMAScript 2015 (ES6), a palavra-chave **var** era utilizada para declarar uma variável. Entretanto, era possível declararmos duas variáveis com o mesmo nome, o que, às vezes, poderia levar a um resultado imprevisível e incorreto.

```
var framework = 'Vue.js';
var framework = 'Angular';
alert(framework);
```

Nesse exemplo, temos duas variáveis com o nome *framework* e a saída é Angular, que foi a última declaração da variável.

A partir da ES6, foi introduzida a palavra-chave **let**, em substituição a **var**:

```
1. let framework = 'Vue.js';
2. let framework = 'Angular'; //ocorre um erro
3. alert(framework);
```

A linha 2 lançará um erro, pois a variável *framework* já fora declarada anteriormente. A ES6 também introduziu a palavra-chave **const**, para dados que têm valor somente de leitura, ou seja, cujo valor não pode ser alterado.

```
const pi = 3.14;
pi = 3.14; // provocará erro
pi = 2 * pi; //também errado
let ang = 2 * pi; //permitido porque ang é uma variável
```

Como Trabalhar com Dados

Imagine que, por exemplo, precisarmos alterar um dado ou atribuir o seu valor a uma outra variável. Para isso, vamos utilizar os operadores de atribuição e aritméticos, que são utilizados para manipular dados numéricos.

Comecemos pelo operador de atribuição (=):

```
1. let x = 10;
2. let y = x; // y = 10
3. x = 15; // agora x = 15
4. let soma = x + y; // soma armazena o valor 25
```

No Quadro 2, vamos detalhar o que está acontecendo nesse Código:

Quadro 2 – Exemplo comentado

Linha(s)	Descrição
Linha 1	Definimos a variável x, que recebe o valor 15.
Linha 2	Definimos a variável y, que recebe o conteúdo da variável x.
Linha 3	A variável x recebe o valor 15.
Linha 4	Definimos a variável soma, que recebe o resultado de x + y, ou, 10 + 15.

Nesse exemplo, você deve ter percebido que utilizamos o operador de adição (**+**). No Quadro 3, temos os demais operadores aritméticos. Para fins de exemplo, considere a variável **x** declarada anteriormente, que armazena o valor 15.

Quadro 3 – Operadores aritméticos

Operador	Descrição	Exemplo	Resultado
+	Adição	z = x + 10	z = 25
-	Subtração	z = x - 10	z= 5
*	Multiplicação	z = x * 10	z = 150
/	Divisão	z = x / 5	z = 3
**	Expoente	z = x **2	z = 225
%	Módulo (resto da operação de divisão)	z = x % 7	z = 1
++	Incremento	z = x++	z = 16
--	Decremento	z = x--	z = 14

Como Manipular Textos

Strings são sequências de caracteres, que formam um texto, e que podem ser representadas de três formas, em JS:

- Aspas duplas. Ex.: “algum texto”;
- Aspas simples. Ex.: ‘algum texto’;
- Acento grave. Ex.: `algum texto`.

Não há diferença entre as aspas duplas e simples, mas há diferença com o acento grave, pois ele permite as chamadas ***templates String.***

Leitura

Os ***templates String*** ou *templates* literais constituem um recurso interessante, em substituição à concatenação de valores.

Quando trabalhamos com texto, o operador de adição (**+**) é utilizado para concatenar ***Strings***.

Veja o exemplo a seguir:

```
let x = “Bom”;
let y = “dia”;
let z = x + y; // o resultado será: z=Bom dia
alert(z);
```

Tenha cuidado ao utilizar o operador de adição (**+**) com números e textos, pois o resultado será sempre um texto:

```
let x = 4;
let y = “a”;
let z = x + y; // o resultado será z=”4a”

let x = “4”;
let y = “4”;
let z = x + y; // o resultado será z=”44”
```

Conversão de Tipos

Como vimos, não é necessário declarar os tipos das variáveis, pois, o *JavaScript* faz a tipagem dinâmica do dado que está trabalhando. Entretanto, toda a entrada de dados feita pelo usuário é sempre considerada uma ***String*** (texto), mesmo que a entrada sejam algarismos, como as notas de um aluno, por exemplo.

Isso pode gerar um resultado incorreto ou indesejado, se considerarmos, por exemplo, que o operador de adição (**+**) tem comportamentos diferentes quando utilizado em texto e ou número. Numa operação na qual exista um número e um texto, o número sempre será convertido para texto e a operação ocorre como se todos os dados fossem ***String***.

Dessa forma, caso seja necessário realizar alguma operação matemática com os valores lidos por meio de uma caixa de diálogo *prompt* ou um formulário com campos de *input*, precisamos, inicialmente, convertê-los em números. Para isso, temos duas funções em *JavaScript* que convertem para número inteiro (**int**) ou decimal (***float***).

Para converter um texto em número inteiro utilizando a função **parseInt**, usamos a sintaxe:

```
variável = parseInt(valor);
```

Para converter um texto em número decimal utilizando a função **parseFloat**, usamos a sintaxe:

```
variável = parseFloat(valor);
```

No próximo exemplo, vamos solicitar ao usuário que digite dois números, por meio de uma caixa de diálogo *prompt*, e exibimos a soma desses dois números:

```
//modo 1: utilizamos uma variável aux para a leitura do valor
let aux = prompt("Digite o valor de a","");
let a = parseInt(aux);
//modo 2: fazemos a leitura do valor e a conversão
let b = parseInt(prompt("Digite o valor de b",""));
//mostra o cálculo
alert(a + " + " + b + " = " + (a+b));
```

Antes de finalizarmos este capítulo, você percebeu que, nos exemplos, utilizamos comentários para destacar alguma informação importante do código? Pois

bem. Sempre é uma boa prática inserirmos comentários em nossos códigos e, em *JavaScript*, podemos fazer isso de duas formas:

Para comentários de várias linhas, utilizamos **/*** comentário ***/**:

```
/* Inserimos aqui nossos comentários
   em várias linhas
*/
```

Para comentários de uma única linha, utilizamos **//**comentário:

```
//Aqui inserimos somente uma linha de comentário
```

Comandos e Funções

Em *JavaScript*, uma função é um bloco de códigos destinado a realizar alguma tarefa específica. Uma função pode ser executada por meio de uma chamada direta ou por meio de um evento.

A sintaxe de definição (ou declaração) de uma função, é apresentada na Figura 1:

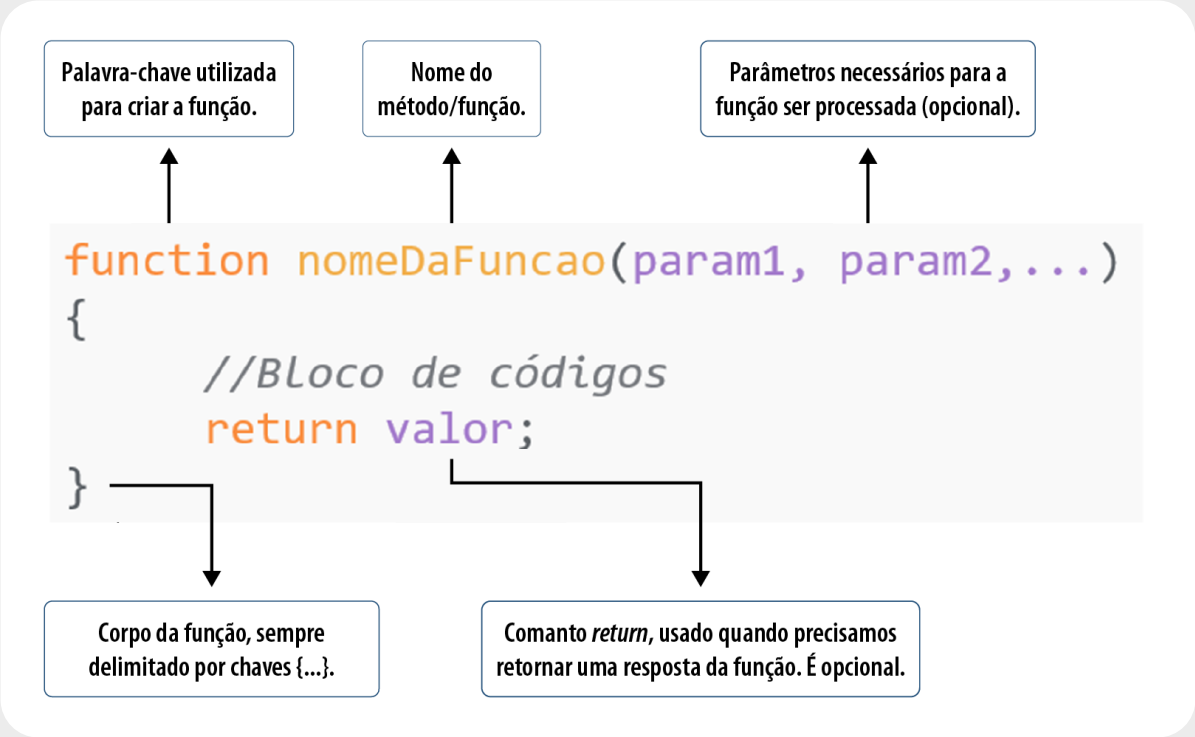


Figura 1 – Sintaxe de uma função

Fonte: Acervo do Conteudista

#ParaTodosVerem: imagem que descreve a sintaxe de uma função *JavaScript*. No centro, há um retângulo com fundo cinza e, em seu interior, a definição da função. Acima desse retângulo, há três retângulos menores e abaixo outros dois, todos contendo a descrição de cada um dos itens dessa sintaxe. Fim da descrição.

Os parâmetros devem ser separados por vírgula e constituem os valores que serão recebidos pela função quando ela é chamada. Dentro da função, os parâmetros comportam-se como variáveis locais, ou seja, somente existem no bloco em que foram declaradas.

Uma função pode ou não retornar uma resposta para o ponto de chamada. Caso necessite de retorno, utilizaremos o comando ***return***. Quando não retorna resultado, a função tem comportamento de procedimento e, nesse caso, não utilizaremos o comando *return* (ao encerrar, a função retornará o valor ***undefined***).

O exemplo a seguir define uma função simples chamada soma, que recebe dois números e retorna a soma deles:

```
function soma(num1, num2){
  return num1 + num2;
}
```

A partir da versão ES6, foi introduzido o formato ***arrow function*** para criar funções mais otimizadas. A sintaxe de uma ***arrow function*** contém a seguinte definição:

```
let identificador = (param1, param2, ...) => {
        //bloco de instruções
}
```

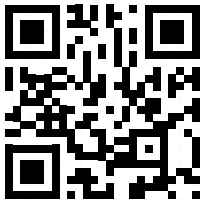
Vejamos o exemplo anterior (soma de dois números) no formato ***arrow function***:

```
let soma = (num1, num2) => { return num1 + num2 };
```

Bem mais simples, não é mesmo?

Leitura

Explore como as ***arrow functions*** permitem uma sintaxe reduzida, mais curta, para funções.



Introdução às Decisões em *JavaScript*

As decisões são importantes em programação, pois permitem executar diferentes ações com base em alguma expressão condicional. A Figura 2 mostra o fluxograma de um programa que lê a média de um aluno e imprime o resultado ("Aprovado" ou "Reprovado"), a partir de uma tomada de decisão (media >= 6.0).

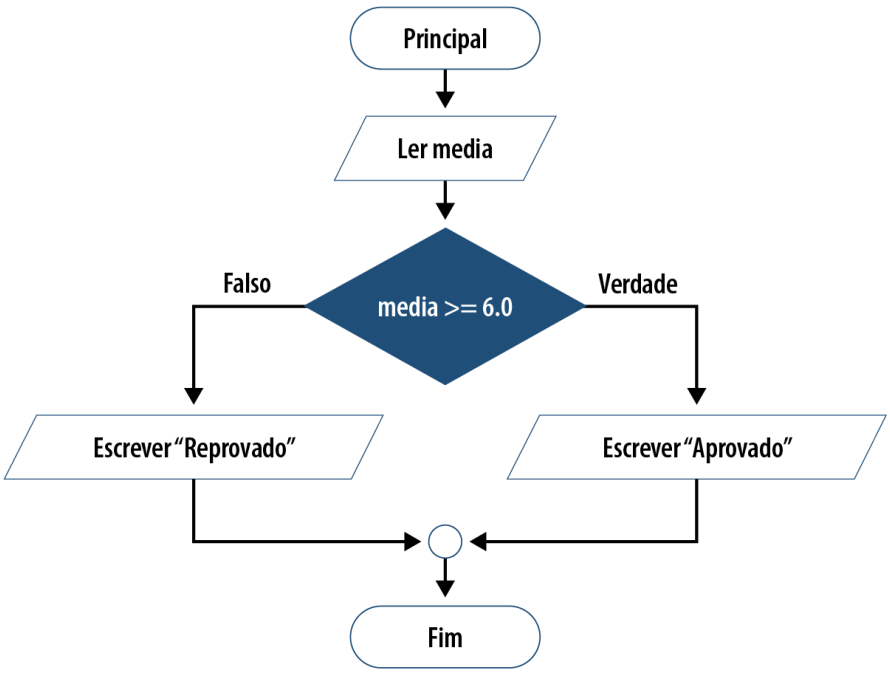


Figura 2 – Tomada de decisões em programação

Fonte: Acervo do Conteudista

#ParaTodosVerem: imagem do fluxograma de uma estrutura de decisão. O fluxograma é composto por figuras geométricas, que são utilizadas para definir os passos e a sequência da estrutura. Nessa Figura, o losango é utilizado para representar a tomada de decisão quanto à média de um aluno ser maior ou igual a seis. Fim da descrição.

Em programação, as decisões precisam ser bem definidas, ou seja, não podem ser ambíguas. Para isso, precisamos tomar decisões que serão construídas a partir de expressões que utilizam operadores relacionais e lógicos.

Os operadores relacionais, mostrados no Quadro 4, são utilizados para comparações, como igualdades ou desigualdades. O resultado de uma comparação será sempre um valor lógico (verdadeiro – ***true***, ou falso – **false**).

Quadro 4 – Operadores relacionais

Operador	Nome	Exemplo	Resultado
==	Igual	a == b	Verdadeiro se a for igual a b.
!=	Diferente	a != b	Verdadeiro se a não for igual a b.
===	Idêntico	a === b	Verdadeiro se a for igual a b e for do mesmo tipo.
<	Menor que	a < b	Verdadeiro se a for menor que b.
>	Maior que	a > b	Verdadeiro se a for maior que b.
<=	Menor ou igual	a <= b	Verdadeiro se a for menor ou igual a b.
>=	Maior ou igual	a >= b	Verdadeiro se a for maior ou igual a b.

Os operadores lógicos, mostrados no Quadro 5, são utilizados para expressões que contenham mais de uma condição ou comparação. Para os exemplos, considere que **a = 3** e **b = 5**.

Quadro 5 – Operadores lógicos

Operador	Descrição	Exemplo

&&	**E** lógico: retorna verdadeiro se ambas as expressões são verdadeira e falso nos demais casos.	a==3 && b<10 //retorna True
		a!=3 && b==5 //retorna False

||	**Ou** lógico: retorna verdadeiro se pelo menos uma das expressões é verdadeira e falso se todas são falsas.	a==3 || b<10 //retorna True
		a!=3 || b==5 //retorna True
		a==1 || b==3 //retorna False

!	**Não** lógico: retorna verdadeiro se o operando é falso e vice-versa.	!(a==3) //retorna False !(a!=3) //retorna True

Estruturas de Decisão

São blocos de código executados somente se uma dada condição for satisfeita, utilizados para controlar o fluxo de execução dos programas. Em JavaScript, temos as seguintes estruturas de decisões:

- **Decisão simples:** *if;*
- **Decisão composta:** *if... else;*
- **Decisão encadeada:** *if... else if... else...;*
- **Estrutura de decisão:** *switch... case.*

A decisão simples (***if***) permite avaliar se o resultado de uma expressão lógica é verdadeiro (***true***), e executa o bloco de códigos inserido nela. Veja a sintaxe e um exemplo:

```
if (condição){
      //bloco de instruções caso a condição seja verdadeira
   }
```

```
let data_hora = new Date();
   let hora = data_hora.getHours();
   if (hora < 12){
      alert("Bom dia!!!");
   }
```

A decisão composta (***if... else***) avalia uma expressão lógica e, caso o resultado seja verdadeiro (***true***), executa o primeiro bloco de Códigos. Caso contrário, executa o segundo bloco de comandos (***else***).

Veja a sintaxe e um exemplo:

```
 if (condição){
     //bloco de instruções caso a condição seja verdadeira
}
else {
    //bloco de instruções caso a condição seja falsa
}
```

```
 let data_hora = new Date();
let hora = data_hora.getHours();
if(hora < 12){
  alert("Bom dia!!!");
}
else{
  alert("Tenha uma ótima tarde/noite");
}
```

A estrutura encadeada (***if... else if... else...***) é utilizada quando temos várias condições que devem ser testadas. Sua sintaxe é exibida a seguir:

```
if(condição 1){
   //bloco de instruções 1
}
else {
   if(condição 2){
      //bloco de instruções 2
   }
   else {
      //bloco de instruções 3
   }
}
```

O exemplo completo, utilizando o condicional encadeado, fica assim:

```
let data_hora = new Date();
let hora = data_hora.getHours();
if (hora < 12){
    alert("Bom dia!!!");
}
else {
    if(hora >= 12 && hora < 18){
        alert("Tenha uma ótima tarde");
    }
    else {
        alert("Tenha uma ótima noite");
    }
  }
```

A estrutura ***switch... case*** é utilizada quando temos várias condições simples, conforme a sintaxe:

```
switch (valor) {
   case valor1:
      //instruções 1
      break;
   case valor2:
      //instruções 2
      break;
   case valor3:
      //instruções 3
      break;
   default:
      //instruções padrão
  }
```

Observações importantes quanto ao uso dessa estrutura:

- Para cada caso, devemos colocar o comando ***break***, que irá finalizar o caso e evitar que o caso posterior seja executado;
- O ***default*** é opcional. Ele é executado quando nenhum caso anterior é acionado;
- Os valores podem ser ***String***, inteiros, caracteres ou reais;
- Não podemos fazer comparações no *case*, como x > y, por exemplo.

Um exemplo completo do uso é apresentado a seguir:

```
let data_hora = new Date();
let dia_semana = data_hora.getDay();
switch (dia_semana){
   case 0: alert("Domingo de descanso merecido.");
      break;
   case 5: alert("Obaaa, sexta-feira.");
      break;
   case 6: alert("Maravilha, sabadão!!");
      break;
   default: alert("Semana longaaaa.");
}
```

Repetindo Códigos

Muitas vezes, precisamos concluir de modo rápido e eficiente tarefas repetitivas, como uma tabuada, por exemplo. Para isso, as Linguagens de Programação têm as estruturas de repetição ou laços de repetição (*loop*).

Essas estruturas são utilizadas quando necessitamos repetir um bloco de instruções um número específico de vezes ou enquanto uma condição for verdadeira. Em *JavaScript* veremos as estruturas ***for*** e ***while***.

Estrutura de Repetição *for*

É compacta, pois inicialização, condição e atualização estão reunidas na declaração do laço. Utilizamos essa estrutura quando sabemos exatamente o número de repetições que serão feitas.

Sua sintaxe é:

```
for (valor inicial; condição; incremento/decremento){
//instruções que serão repetidas
}
```

Vejamos um exemplo que conta de 0 até 9 e exibe um ***alert*** na tela com o valor da variável **cont**:

```
let cont;
for (cont = 0; cont < 10 ; cont++){
   alert("Número: " + cont + "<br />");
}
```

Algumas observações sobre o uso do ***for*** são:

- A estrutura utiliza uma variável inteira para controlar o número de vezes que deve ser repetida a execução das instruções;
- É inicializada com um valor qualquer e é incrementada ou decrementada a cada iteração;
- A condição, geralmente, verifica se essa variável já chegou a determinado valor para decidir se o laço deve ser encerrado.

Estrutura de Repetição *while*

Executa um bloco de instruções enquanto certa condição for verdadeira.

Sua sintaxe é:

```
//inicialização da variável de controle
while(condição){
    //bloco de instruções
    //atualização da variável de controle
}
```

O exemplo a seguir conta de 0 até 9 e exibe um ***alert*** na tela com o valor da variável **cont**:

```
let cont = 0;
while(cont < 10){
    alert("Número: " + cont + "<br />");
    cont = cont + 1; //ou cont++;
}
```

Algumas observações sobre o uso da estrutura ***while*** são:

- A variável de controle deve ser inicializada antes do bloco;
- No interior do laço, é inserida uma instrução que atualiza a variável de controle. Com isso, em algum momento, a condição se tornará falsa.

Parando um Laço

Podemos usar comando ***break*** para parar um determinado laço de repetição conforme alguma condição no nosso Código. No exemplo a seguir, vamos somar uma quantidade indeterminada de valores digitados pelo usuário. Como não sabemos exatamente a quantidade, utilizamos uma condição ***if*** para parar o laço. Assim, quando o usuário pressionar a tecla ***enter*** sem digitar um valor, vamos encerrar o laço e exibir o resultado da soma:

```
let soma = 0;
while (true){
        let num = Number(prompt("Enter a number", "));
    if(!num)
       break;
    soma += num;
}
alert("Soma = " + soma);
```

Continuando um Laço

O comando ***continue*** interrompe a iteração atual e força o laço a continuar para a próxima iteração. No exemplo a seguir, desejamos imprimir somente os números ímpares de 0 até 10. Portanto, sempre que o número for par, o condicional ***if*** executa o comando ***continue*** e pula para a próxima iteração, ignorando a impressão do número par:

```
for (let i = 0; i < 10; i++) {
  if (i % 2 == 0)
     continue;
  alert(i); // imprime somente os ímpares de 0 a 10
}
```

Vetores

Vetores podem ser definidos como uma coleção de elementos ou itens que respondem a um mesmo nome e que podem ser acessados segundo a posição (índice) que ocupam dentro do vetor.

Na Figura 3, temos a representação gráfica de um vetor:

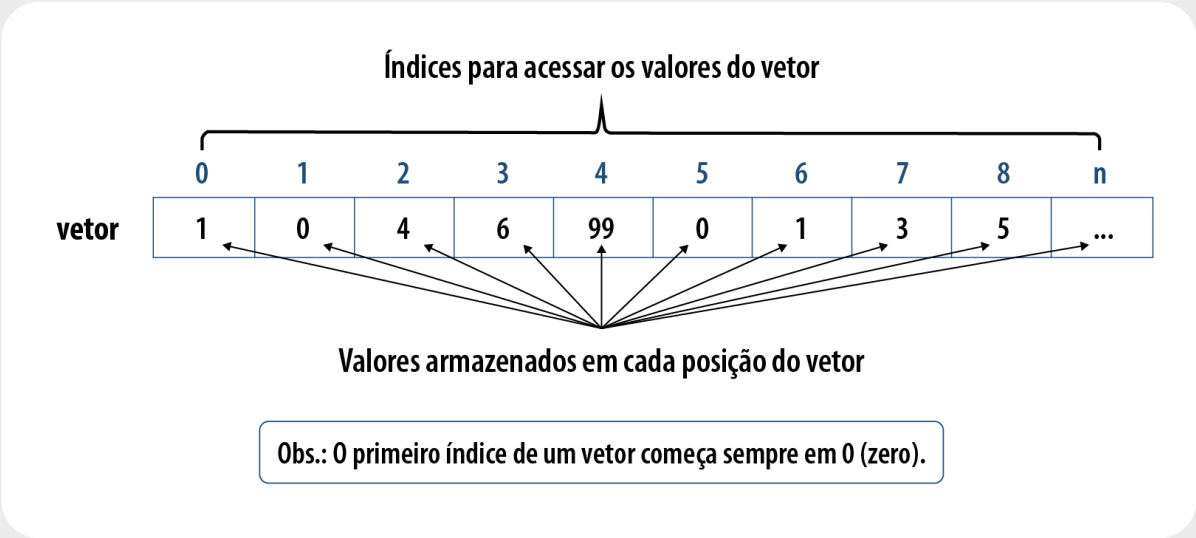


Figura 3 – Representação gráfica de um vetor

Fonte: Acervo do Conteudista

#ParaTodosVerem: imagem da representação gráfica de um vetor. No centro, há um retângulo dividido em 10 partes iguais, contendo números. Acima desse retângulo, há o título "Índices para acessar os valores do vetor". Abaixo do retângulo, há um texto "Valores armazenados em cada posição do vetor" e na parte inferior da imagem, uma observação em destaque: "o primeiro índice de um vetor começa sempre em zero". Fim da descrição.

Para criar um vetor em *JavaScript*, utilizamos o objeto ***Array***. Esse objeto, além de criar um vetor em memória, também nos fornece diversos métodos para manipular o vetor criado.

Vejamos um exemplo com alguns vetores criados em *JavaScript*:

```
//vetor sem tamanho definido, inicialmente vazio:
var vetorA = new Array();
var vetorB = [ ];
//vetor com posições e valores definidos:
var vetorC = new Array("maça", "banana", "morango");
var vetorD = [ "maça", "banana", "morango" ];
var vetorE = [ 12, 6, 21, 5, 7, 209, 3 ];
```

Perceba que, em todos os casos anteriores, podemos inserir mais posições sempre que necessário. O Vetor não tem um tamanho fixo.

Podemos visualizar ou atribuir um valor para um vetor, acessando cada posição por meio do índice, que deve ficar entre colchetes **[...]**.

Veja o exemplo a seguir:

```
let frutas = [];
frutas[0]= “maça”;
frutas[1]= “banana”;
frutas[2]= “morango”;
alert(frutas[2]); //Imprime na tela “morango”

let data_hora = new Date();
let meses = [“JAN”,”FEV”,”MAR”,”ABR”,”MAI”,”JUN”,”JUL”,”AGO”,”SE-
T”,”OUT”,”NOV”,”DEZ”];
alert(meses[data_hora.getMonth()]); //Imprime na tela o mês atual
```

É bastante comum, quando trabalhamos com vetores, utilizarmos uma estrutura de repetição para acessar suas posições. A mais utilizada é a estrutura ***for***, porém, nada impede o comando ***while*** seja utilizado. Nos exemplos a seguir, observe a comparação entre o vetor sem o uso de uma estrutura de repetição e com o uso da estrutura ***for***:

```
//Vetor sem o uso da estrutura for
let frutas = [ ];
frutas[0]=”maçã”;
frutas[1]=”banana”;
frutas[2]=”morango”;
console.log(frutas[0]);
console.log(frutas[1]);
console.log(frutas[2]);
```

```
//Vetor com o uso de repetições
let i;
let frutas = [ ];
frutas[0]=”maçã”;
frutas[1]=”banana”;
frutas[2]=”morango”;
for (i=0; i<=2; i++){
    alert(frutas[i]);
}
```

É importante notar como o uso correto da estrutura de repetição torna mais fácil percorrer um vetor para consultas ou atualizações. Porém, o exemplo acima funciona muito bem para um vetor de somente três elementos! Na prática, é comum que o tamanho do vetor seja desconhecido do programador!

Isso poderia constituir um problema, porém, podemos utilizar a propriedade ***length***, que retorna o número de elementos de um vetor.

Para finalizarmos esta Unidade, vamos lembrar-nos de que nosso vetor é dinâmico e, portanto, permite que alteremos um valor ou até mesmo façamos a inclusão de um novo elemento. No último caso, estamos falando do método ***push(),*** utilizado para adicionar um novo item ao final do vetor.

No exemplo final, temos um vetor contendo três fabricantes de carros. Nosso objetivo é incluir um novo fabricante e alterar o nome do terceiro item, que está grafado incorretamente (“Hundai”). Finalmente, utilizaremos o laço ***for*** combinado com a propriedade ***length***, para listar todos os elementos do vetor:

```
let carros = [“Honda”, “Fiat”, “Hundai”];
carros[2] = “Hyundai”; //Substitui o valor do elemento na posição 2
let c = prompt(“Digite um fabricante: “);
carros.push(c); //Adiciona um novo elemento ao final do vetor

//A propriedade lenght retorna o tamanho do vetor
for(let i = 0; i < carros.length; i++){
    alert(carros[i]);
}
```

Veja, a seguir, no Quadro 6 alguns dos principais métodos utilizados com vetores em *JavaScript*.

Considere o vetor vogais = [“a”,”e”,”i”]:

Quadro 6 – Métodos com vetores

Exemplo	Descrição	Resultado
vogais.push(“o”)	Adiciona um elemento ao final do vetor.	[“a”,”e”,”i”,”o”]
vogais.pop()	Remove o último elemento do vetor.	[“a”,”e”]
vogais.indexOf(“i”)	Retorna o primeiro índice em que o elemento pode ser encontrado no vetor, ou -1, caso contrário.	2
vogais.includes(“e”)	Retorna *true* se o elemento está contido no vetor.	*True*
vogais.reverse()	Inverte a ordem dos elementos do vetor.	[“i”,”e”,”a”]

Leitura

Visite a referência completa sobre vetores, propriedades e métodos em *JavaScript*.

MATERIAL COMPLEMENTAR

Sites

Codepen
Ferramenta *on-line* para testar e exibir trechos de código HTML, CSS e *JavaScript* que funciona como um editor *on-line* e ambiente de aprendizagem.
https://bit.ly/3re2C3R

JSFiddle
Plataforma *on-line* que permite criar, testar e exibir trechos de códigos em HTML, CSS e JS.
https://bit.ly/3rcmtAz

Leituras

W3school
Site voltado à aprendizagem de conteúdos de Tecnologias como HTML, CSS, *Javascript* e *Python*, entre outros. Visite o tutorial de *JavaScript*.
https://bit.ly/3EwRF0z

30 Dias de *JavaScript*
A matéria apresenta uma série de desafios para melhorar as habilidades de codificação por meio de mini projetos diários, utilizando programação *Web*.
https://bit.ly/3PB1IYR

REFERÊNCIAS BIBLIOGRÁFICAS

CLARK, R. *et al*. **Introdução ao HTML5 e CSS3:** a evolução da *web*. Rio de Janeiro: Alta *Books*, 2014.

SILVA, M. S. **Construindo *sites* com CSS e (X) HTML:** *sites* controlados por folhas de estilo em cascata. São Paulo: Novatec, 2008.

SILVA, M. S. **CSS3:** Desenvolva aplicações *web* profissionais com uso dos poderosos recursos de estilização das CSS3. São Paulo: Novatec, 2012.

TERUEL, E. C. **HTML 5:** guia prático. 2. ed. São Paulo: Erica, 2013.

***WEB Standards.* W3C**. Disponível em: <https://www.w3.org/standards/webdesign/script>. Acesso em: 06/03/2023.